# Express.js Workshop - NodeSummit 2018

## Express

## Learning Objectives

- Learn how to create HTTP servers with Express
- Understanding Middleware
- Security

## Express Server Example

### 01_hello_world.js:

```
'use strict'

const express = require('express')
const app = express()

app.get('/', (req, res) => {
  res.send('Hello World!')
})

app.listen(3000, () => {
  console.log('Example app listening on port 3000!')
})
```

## Routing Methods

- checkout
- copy
- delete
- get
- head
- lock
- merge
- mkactivity

## Routing Methods

- mkcol
- move
- m-search
- notify
- options
- patch
- post
- purge

## Routing Methods

- put
- report
- search
- subscribe
- trace
- unlock
- unsubscribe

## Dynamic Routing

### 02_routes.js:

```
'use strict'

const express = require('express')
const app = express()

app.get('/user/:name', (req, res) => {
  const { name } = req.params
  res.send('Hello ' + name)
})

app.listen(3000, () => {
  console.log('Example app listening on port 3000!')
})
```

## Route Handler

### 03_handler.js:

```
'use strict'

const express = require('express')
const app = express()

app.get('/user/:name', (req, res, next) => {
  const { name } = req.params
  res.send('Hello ' + name)
})

app.listen(3000, () => {
  console.log('Example app listening on port 3000!')
})
```

# Middleware

### 04_middleware.js:

```javascript
'use strict'

const express = require('express')
const app = express()

app.use((req, res, next) => {
  if (req.url === '/') return next()
  else return next(new Error('Not Found'))
})

app.get('/', checkQuery, (req, res) => {
  res.send('Hello World!')
})

app.listen(3000, () => {
  console.log('Example app listening on port 3000!')
})

function checkQuery (req, res, next) {
  if (req.query.name) return next()
```

## Security

Encode all untrusted data

## Security - Backend

Backend: escape-html

Note: When using the escaped value within a tag, it is only suitable as the value of an attribute, where the value is quoted with either a double quote character (") or a single quote character (').

## Security - CSS Encoding

- Front-end: CSS.escape Web API or the CSS.escape polyfill
- Backend: CSS.escape package (same as the polyfill above)

# Security - JavaScript Encoding

- Front-end: js-string-escape - This is a back-end Node module, but can also be used on the front-end.
- Backend: js-string-escape

## Security - URL and URI Encoding

- Frontend:

  encodeURICompnent()

- Back-end: urlencode

To read a bit more about the high value of encoding user input, take a look at the XSS Prevention Cheat Sheet by OWASP.

## Prevent Parameter Pollution to Stop Possible Uncaught Exceptions

`curl http://example.com:8080/endpoint?name=Itchy&name=Scratchy`

```
app.get('/endpoint', (req, res) => {
  if (req.query.name) {
    res.status(200).send('Hi ' + req.query.name.toUpperCase())
  } else {
    res.status(200).send('Hi')
  }
})
```

## Add Helmet to Set Sane Defaults

```
const express = require('express')
const helmet = require('helmet')

const app = express()

app.use(helmet())
```

## Tighten Session Cookies

- secret - A secret string for the cookie to be salted with.
- key: The name of the cookie - if left default (connect.sid), it can be detected and give away that an application is using Express as a web server.
- httpOnly - Flags cookies to be accessible by the issuing web server, which assists in preventing session hijacking.
- secure - Ensure that it is set to true - which requires TLS/SSL - to allow the cookie to only be used with HTTPS requests, and not insecure HTTP requests.
- domain - Indicates the specific domain that the cookie can be accessed from.
- path - indicates the path that the cookie is accepted on within an application's domain.
- expires - The expiration date of the cookie being set. Defaults to a session cookie. When setting a cookie, the application is storing data on the server. If a timely expiration is not set up on the cookie, the Express application could start consuming resources that would otherwise be free.

## Block Cross-Site Request Forgeries

```
const express = require('express')
const csrf = require('csurf')

const app = express()

app.use(csrf())

app.use((req, res, next) => {
 // Expose variable to templates via locals
 res.locals.csrftoken = req.csrfToken()
 next()
})
```

## Block Cross-Site Request Forgeries

```
<input type="hidden" name="_csrf" value={{csrftoken}} />
```

## Don't Use Evil Regular Expressions

### EVIL REGEX PATTERNS CONTAINS:

- Grouping with repetition
- Inside the repeated group:
- Repetition
- Alternation with overlapping

## EXAMPLES OF EVIL PATTERNS:

- (a+)+
- ([a-zA-Z]+)*
- (a|aa)+
- (a|a?)+
- (.*a){x} | for x > 10

All the above are susceptible to the input aaaaaaaaaaaaaaaaaaaaaaaa! (The minimum input length might change slightly, when using faster or slower machines).

## Add Rate Limiting

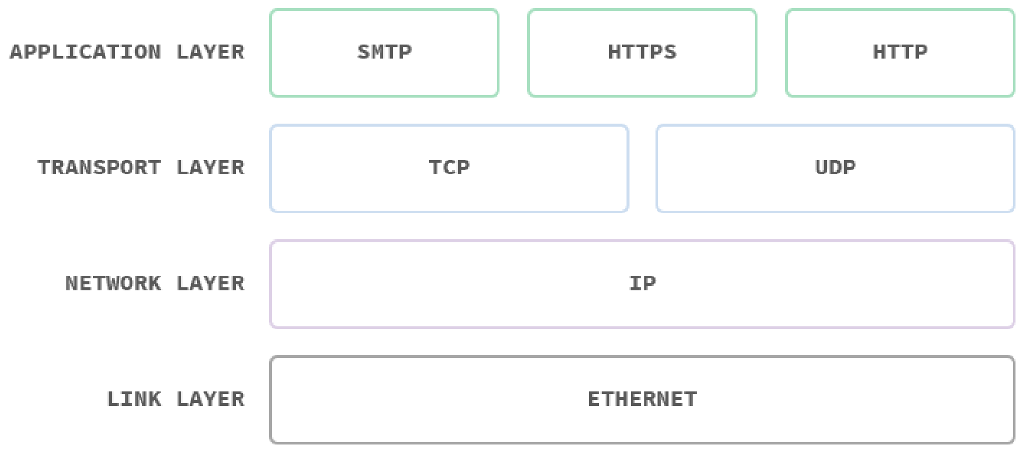
```
const express = require('express')
const redis = require('redis')

const redisClient = redis.createClient()
const app = express()

const limiter = require('express-limiter')(app, redisClient);

// Limit requests to 100 per hour per ip address.
limiter({
  lookup: ['connection.remoteAddress'],
  total: 100,
  expire: 1000 * 60 * 60
})
```

Networking Basics

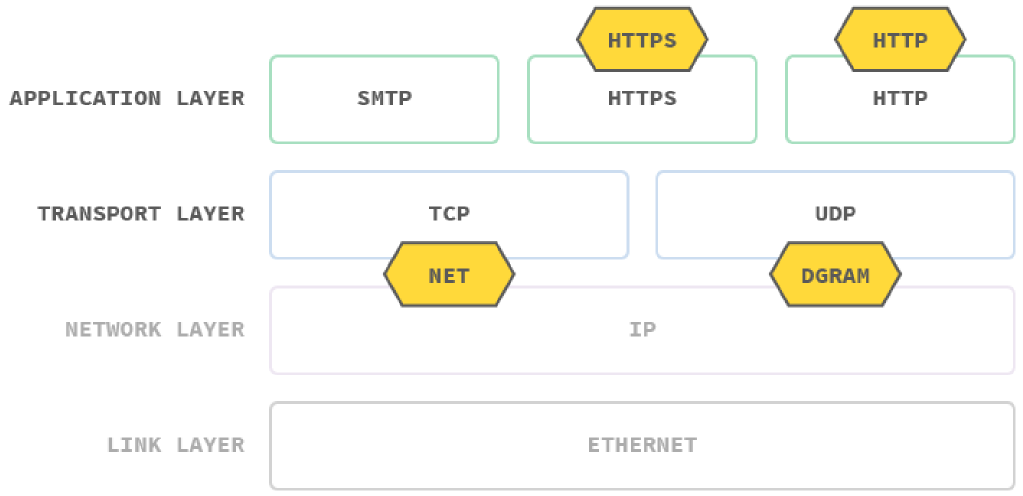
## Learning Objectives

- Understand the basics of TCP, HTTP and HTTPS
- Learn how Node.js exposes each of these via built-in modules

# Network Stack



# Network Stack

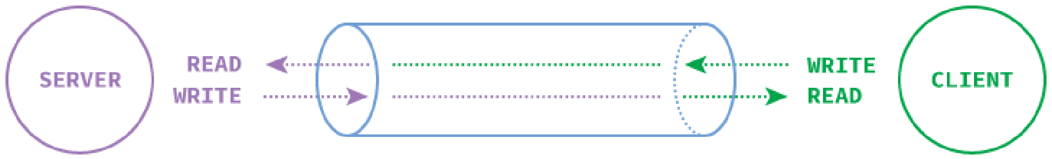


## TCP

**T**ransmission **C**ontrol **P**rotocol

- Connection oriented
- Guarantees lossless and ordered transmission of data
- Implemented by Node.js core `'net'` module: http://nodejs.org/api/net.html

# TCP



## TCP Server

### 01_tcp_server.js:

```
'use strict'

const net = require('net')
const server = net.createServer()
const PORT = 8000

server
  .on('connection', onConnection)
  .on('listening', onListening)
  .listen(PORT)

function onConnection (conn) {
  conn.write('You are in a huge cave\r\n')
  conn.pipe(conn)
}

function onListening () {
  console.log('TCP server listening on port', PORT)
}
```

## TCP Server

### 01_tcp_server.js:

```
'use strict'

const net = require('net')
const server = net.createServer()
const PORT = 8000

server
  .on('connection', onConnection)
  .on('listening', onListening)
  .listen(PORT)

function onConnection (conn) {
  conn.write('You are in a huge cave\r\n')
  conn.pipe(conn)
}

function onListening () {
  console.log('TCP server listening on port', PORT)
}
```

Using the Node.js `'net'` module to create a TCP server

## TCP Server

### 01_tcp_server.js:

```
'use strict'

const net = require('net')
const server = net.createServer()
const PORT = 8000

server
  .on('connection', onConnection)
  .on('listening', onListening)
  .listen(PORT)

function onConnection (conn) {
  conn.write('You are in a huge cave\r\n')
  conn.pipe(conn)
}

function onListening () {
  console.log('TCP server listening on port', PORT)
}
```

- Registering callbacks with server and listening on given `PORT`
- `onConnection` invoked for each new client connection
- `onListening` only invoked once

## TCP Server

### 01_tcp_server.js:

```
'use strict'

const net = require('net')
const server = net.createServer()
const PORT = 8000

server
  .on('connection', onConnection)
  .on('listening', onListening)
  .listen(PORT)

function onConnection (conn) {
  conn.write('You are in a huge cave\r\n')
  conn.pipe(conn)
}

function onListening () {
  console.log('TCP server listening on port', PORT)
}
```

- Send invitation message to client by writing to TCP socket connection

## TCP Server

### 01_tcp_server.js:

```
'use strict'

const net = require('net')
const server = net.createServer()
const PORT = 8000

server
  .on('connection', onConnection)
  .on('listening', onListening)
  .listen(PORT)

function onConnection (conn) {
  conn.write('You are in a huge cave\r\n')
  conn.pipe(conn)
}

function onListening () {
  console.log('TCP server listening on port', PORT)
}
```

- Piping all data sent to server back to client causes all client messages to be echoed back to the client, like an "echo server"

## TCP Client

### 02_tcp_client.js:

```
'use strict'

const PORT = 8000
const net = require('net')
const client = net.connect(PORT)

client.on('data', onData)

function onData (data) {
  process.stdout.write('server: ' + data.toString())
  setTimeout(respond, 1000)
}

function respond () {
  const msg = 'Describe cave\r\n'
  process.stdout.write('client: ' + msg)
  client.write(msg)
}
```

## TCP Client

### 02_tcp_client.js:

```
'use strict'

const PORT = 8000
const net = require('net')
const client = net.connect(PORT)

client.on('data', onData)

function onData (data) {
  process.stdout.write('server: ' + data.toString())
  setTimeout(respond, 1000)
}

function respond () {
  const msg = 'Describe cave\r\n'
  process.stdout.write('client: ' + msg)
  client.write(msg)
}
```

# TCP Client

## 02_tcp_client.js:

```
'use strict'

const PORT = 8000
const net = require('net')
const client = net.connect(PORT)

client.on('data', onData)

function onData (data) {
  process.stdout.write('server: ' + data.toString())
  setTimeout(respond, 1000)
}

function respond () {
  const msg = 'Describe cave\r\n'
  process.stdout.write('client: ' + msg)
  client.write(msg)
}
```

- Registering the `onData` callback which will be invoked every time the server sends a message

## TCP Client

### 02_tcp_client.js:

```
'use strict'

const PORT = 8000
const net = require('net')
const client = net.connect(PORT)

client.on('data', onData)

function onData (data) {
  process.stdout.write('server: ' + data.toString())
  setTimeout(respond, 1000)
}

function respond () {
  const msg = 'Describe cave\r\n'
  process.stdout.write('client: ' + msg)
  client.write(msg)
}
```

- Logging server message and scheduling response

## TCP Client

### 02_tcp_client.js:

```
'use strict'

const PORT = 8000
const net = require('net')
const client = net.connect(PORT)

client.on('data', onData)

function onData (data) {
  process.stdout.write('server: ' + data.toString())
  setTimeout(respond, 1000)
}

function respond () {
  const msg = 'Describe cave\r\n'
  process.stdout.write('client: ' + msg)
  client.write(msg)
}
```

- Logging the response we are about to send and sending it by *writing* to the TCP socket

## TCP Connections

### End TCP Client Connection

```
client.end()
```

- Terminates the *client* part of a connection
- You may still get data events on the *server*

# TCP Connections

## Close TCP Server

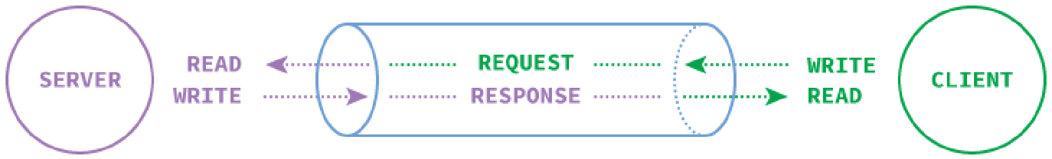
```
server.close()
```

- The *server* accepts no more *client* connections, but keeps existing ones

## HTTP

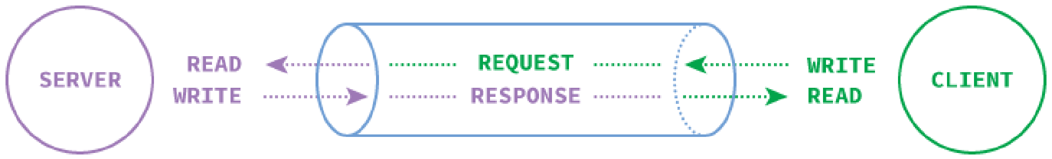
- **Application layer** protocol
- Request / response based
- Sits on top of a **transport layer** protocol, like TCP or UDP

## HTTP Request & Response



- Abstractions of the TCP socket
- **Request**:
  - `http.IncomingMessage` is a *readable stream*
  - Represents the part of the socket that is *readable* to the server and *writable* by the client

# HTTP Request & Response



- Abstractions of the TCP socket
- **Request**:
    - `http.IncomingMessage` is a *readable stream*
    - Represents the part of the socket that is *readable* to the server and *writable* by the client
- **Response**:
    - `http.ServerResponse` is a *writable stream*
    - Represents the part of the socket that is *writable* by the server and *readable* to the client

## HTTP Server

### 03_http_server.js:

```js
'use strict'

const http = require('http')
const server = http.createServer()
const PORT = 8000

server
  .on('request', onRequest)
  .on('listening', onListening)
  .listen(PORT)

function onRequest (req, res) {
  res.writeHead(200, { 'Content-Type': 'text/plain' })
  res.end('You are still in a huge cave\r\n')
}

function onListening () {
  console.log('HTTP server listening on port', PORT)
}
```

Is very similar to ...

## TCP Server

### 01_tcp_server.js:

```javascript
'use strict'

const net = require('net')
const server = net.createServer()
const PORT = 8000

server
  .on('connection', onConnection)
  .on('listening', onListening)
  .listen(PORT)

function onConnection (conn) {
  conn.write('You are in a huge cave\r\n')
  conn.pipe(conn)
}

function onListening () {
  console.log('TCP server listening on port', PORT)
}
```

# HTTP Server

**03_http_server.js:**

```
'use strict'

const http = require('http')
const server = http.createServer()
const PORT = 8000

server
  .on('request', onRequest)
  .on('listening', onListening)
  .listen(PORT)

function onRequest (req, res) {
  res.writeHead(200, { 'Content-Type': 'text/plain' })
  res.end('You are still in a huge cave\r\n')
}

function onListening () {
  console.log('HTTP server listening on port', PORT)
}
```

Using the Node.js `'http'` module to create an HTTP server

## HTTP Server

### 03_http_server.js:

```
'use strict'

const http = require('http')
const server = http.createServer()
const PORT = 8000

server
  .on('request', onRequest)
  .on('listening', onListening)
  .listen(PORT)

function onRequest (req, res) {
  res.writeHead(200, { 'Content-Type': 'text/plain' })
  res.end('You are still in a huge cave\r\n')
}

function onListening () {
  console.log('HTTP server listening on port', PORT)
}
```

- Registering callbacks with server and listening on given `PORT`
- `onRequest` invoked for *each* client request, `onListening` invoked only once
- Connections are handled for us; an `onConnection` handler is not needed

## HTTP Server

### 03_http_server.js:

```
'use strict'

const http = require('http')
const server = http.createServer()
const PORT = 8000

server
  .on('request', onRequest)
  .on('listening', onListening)
  .listen(PORT)

function onRequest (req, res) {
  res.writeHead(200, { 'Content-Type': 'text/plain' })
  res.end('You are still in a huge cave\r\n')
}

function onListening () {
  console.log('HTTP server listening on port', PORT)
}
```

## HTTPS

- **Application layer** request / response based protocol for secure communications
- Layers the HTTP protocol on top of *TLS/SSL*
- The `'https'` module exposes a very similar API to `'http'` module

## HTTPS

### 04_https_server.js:

```
'use strict'

const PORT = 8000
const https = require('https')
const fs = require('fs')
const server = https.createServer({
  pfx: fs.readFileSync('some_cert.pfx')
})

server
  .on('request', onRequest)
  .on('listening', onListening)
  .listen(PORT)

function onRequest (req, res) {
  res.writeHead(200, { 'Content-Type': 'text/plain' })
  res.end('You are still in a huge, but secure, cave\r\n')
}

function onListening () {
```

## HTTP

### 03_http_server.js:

```
'use strict'

const http = require('http')
const server = http.createServer()
const PORT = 8000

server
  .on('request', onRequest)
  .on('listening', onListening)
  .listen(PORT)

function onRequest (req, res) {
  res.writeHead(200, { 'Content-Type': 'text/plain' })
  res.end('You are still in a huge cave\r\n')
}

function onListening () {
  console.log('HTTP server listening on port', PORT)
}
```

## Summary

- The `'net'` module provides an interface to the TCP layer by exposing an API for servers and clients
- TCP sockets are *duplex* streams
- The `'http'` module implements the HTTP protocol
- The HTTP API `request` and `response` objects abstract the underlying socket, allowing us to set headers and writing data to the stream
- The `'https'` module implements the HTTPS protocol with very similar API to the `'http'` module